

II.5. Algorithm Performance Analysis

The amount of time that it takes a computer to perform a task is primarily a function of the algorithm that is being performed. If an algorithm has many steps in it, this may cause the program to take a long time to execute. More commonly, however, an algorithm will take a long time to execute because of having to go through a loop a large number of times. Nesting one loop within another will increase the execution time considerably more.

The number of times that a program cycles through a loop is almost always related to the “size of the problem”. The size of the problem is usually given by one or two parameters, for example the number of elements that need to be sorted in a sort algorithm. We introduce here a notation, called the “Big-Oh” notation, which describes the asymptotic performance of an algorithm; that is, the performance as the size of the problem increases.

Let n be a non-negative integer parameter describing the size of the problem and let $f(n)$ be a function describing the actual performance of the algorithm. The performance could be in terms of the amount of time it takes to perform the algorithm, or the number of steps that need to be executed multiplied by the number of times they must be re-executed. The relative time it takes to perform different types of operations (e.g. call a function, multiply two numbers) can also be considered using weighting functions.

Let $g(n)$ be another function. If there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$, such that $f(n) \leq cg(n)$ for every integer $n \geq n_0$, then $g(n)$ is the “order” of the algorithm. The order of the algorithm is then written in “Big-Oh” notation as $O(g(n))$. For example, if the time that it takes to execute an algorithm is given by $f(n) = 7n - 2$, then $g(n) = n$ is a valid function for $g(n)$ because, if $c = 7$ and $n_0 = 1$, the criteria is satisfied. The order of the algorithm is therefore written in Big-Oh notation as $O(n)$.

Note that, $g(n) = n^2$ is also a valid function, however, since we always want to show our algorithms in the best light (show them to be as fast as possible), we always choose the smallest function possible. We also always use the simplest function possible, so we would use $g(n) = n$ rather than $g(n) = n - 2$.

Most algorithms fall into one of the following types.

Logarithmic	$O(\log n)$	Very good
Linear	$O(n)$	Good
Quadratic	$O(n^2)$	Ok
Polynomial	$O(n^k) \quad k \geq 1$	Poor if $k > 2$
Exponential	$O(a^n) \quad a > 1$	Awful

In most cases the number of cycles is not only dependent on the size of the problem, but also on the data itself. In these cases, it is possible to use the same concepts to establish minimum, maximum or average performance for an algorithm, using the same techniques.

Creating algorithms with as fast as possible asymptotic behavior is important because the performance of a poor algorithm will not get much better, even if the hardware is improved. For example, consider an algorithm that is $O(2^n)$ and is able to process a problem of size m in 10 seconds. If the hardware speed is improved so that it is 256 times as fast, then only 8 more elements can be processed before it will take 10 seconds again (i.e. now a problem of size $m + 8$ will take 10 seconds). This is not a great improvement.

Some problems have inherent limits in how quickly they can be solved, even if the best possible algorithm is used. This is referred to as the computational complexity of the problem. For example, some problems can be proved mathematically to be only solvable with algorithms that are exponential in the size of the problem. Computer Scientists are therefore limited in what they can do. They can find the best algorithm, but cannot go beyond that. As Engineers, we can re-define the problem, to add more information that could possibly reduce the computational complexity of the problem.